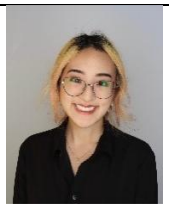
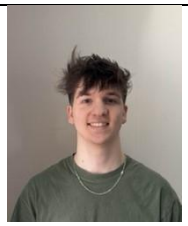
01-T2 - Minecraft Game Mode

Design Document (SDD)

CS 4850 – Section 03 – Spring 2025, February 2nd

Professor Sharon Perry

 Ashley Ahn Design	 Matthew Elledge Programmer	 AnnaGrace Gwee Design	 Bryan Nguyen Team Lead Documentation	 Logan Slicker Programmer

Name	Role	Cell Phone	Alternate Email
Ashley Ahn	Design	(404) 953-1062	Ashleyahn.30@gmail.com
Matthew Elledge	Programming	(404) 207-2916	Mtelledge14@gmail.com
AnnaGrace Gwee	Design	(478) 278-0198	Annagracegwee@gmail.com
Bryan Nguyen	Documentation	(470) 237-9923	Bryann0000@yahoo.com
Logan Slicker	Programming	(404) 983-0209	Lslicker24@gmail.com
Sharon Perry	Project Advisor	(470) 555-1212	Sperry46@kennesaw.edu
Norman Reid	Sponsor	(470) 578-5931	Nreid10@kennesaw.edu

Table of Contents

Introduction

The goal of this project is to complete a functioning game mode for the KSU Minecraft Server. The game mode will be a Tower Defense style game utilizing the style and creatures of Minecraft alongside easy-to-understand interfaces and controls to allow a simple and cohesive game mode for the server's players. Our implementation will also support easy to configure settings and clear documentation to allow future maintenance, maintainability, and expansions for the game mode.

Design Considerations

Assumptions & Dependencies

- Supports Minecraft 1.21 (Latest Version)
- Utilizes Paper API
- Utilizes JetBrainsRuntime Environment
- Utilizes HotSwapAgent

General Constraints

- Must be maintainable
  - Public repository
  - Proper documentation
  - Practical code and comments
- Must be asynchronous (Optimize usage of server resources)
  - File IO (Database)
  - HTTP API Calls
  - Querying
  - Tab Completion
- All in-game/console messages must be configurable
  - Configurable through the config file
  - Supports MiniMessage format
  - In-game tab completion of chat commands
- Bug Testing and playtesting
  - Bug Testing will be in-house
  - Playtesting will be in-house
    - Once the game mode is polished, playtesting will begin from the players
    - Feedback will be obtained from the players

Goals & Guidelines

- Milestone 1 – February 10th
  - Working test environment on PC
  - GitHub Repository set up
  - Research on Paper and Plugin Template completed
  - Game Design Document completed
    - Command Structures considered
    - Libraries solidified
    - Game mechanics and rules
    - World and Map Planning

- Milestone 2 – March 17th
  - Verify asynchronous compatibility
  - Working Prototype
  - Completed World/Map
  - Set-up database connection, table creation, and read/write

- Milestone 3 – April 14th
  - Polish Game Mode
  - Bug Testing
  - Play Testing
    - With participation and feedback of members from the server
  - Implement feedback
  - Documentation completed

Development Methods

The development will follow the template provided by the sponsors, where the project will be coded in Java. The tools JetBrainsRuntime, HotSwapAgent will be utilized to help design the game. The IntelliJ Community IDE will be utilized in conjunction with the IntelliJ Minecraft Development Plugin, allowing for a predesigned template for its namesake. The Paper API will be applied to communicate with the server and utilize the supported tools for plugin development.

Architectural Strategies

Our main language for this plugin will be Java, as that is the language Minecraft is coded in. We considered Kotlin, a language designed by Microsoft, to be a simpler Java, but determined it would be more efficient for us to use Java. Through Java, we will be implementing the Paper API Framework, to interface with the server. Paper is a very common, high-performance framework used in most homebrew Minecraft plugins.

We will likely have to use libraries to extend the functionality of our plugin, to do things like add a GUI or leaderboard. However, most of the design can be done straight from Paper's built in functions. Their event, adventure, and entity APIs allow us to do everything from control mob AI, to spawning entities on the players screen for GUI functionality.

It will also be important to add a proper command structure to each action that is taken by the plugin, as it will increase our efficiency in debugging. These commands will do operations like spawn in any mob of our choosing with whatever health for the level we specify, adding money to the player to buy whatever tower required, round setting and skipping, etc. Also setting leaderboard stats, or commands to display your current time on your attempt will all be useful to make sure we can have any game state possible for testing.

Basic error handling might be necessary. However, the game will not really interface with anything on the client's side. Basic edge case detection will be used for things like spam upgrading towers, making sure any income multipliers are not recursive, or other things to damage player experience. Luckily, since it will be a plugin, we will not have to worry about crashing clients' games, or computers.

Memory management and storage will be important. Ensuring we do not overflow the server with requests, or dead tasks is integral to having an efficient server that hundreds of people can enjoy. We hope to have well scheduled tasks, and weak references for this reason. Opting for larger mob HP pools instead of mass spawning smaller mobs will also be a strategy we will employ, as every entity used in Minecraft has a large minimum overhead. Also designing efficient JSONs for player data that can track all players progress with minimal overhead is very important.

System Architecture

Blocks Tower Defense

Core Layer:

Main class(

Config

Runtime calls

Manages where information is sent)

Game logic layer:

RoundManager()

MobSpawner()

TowerSpawner()

HitDetection()

Win_LossDetection()

TowerAI()

MobAI()

Player Interactions:

CommandListeners()

GUIupdater()

Persistance Layer:

PlayerSaves()

LeaderboardUpdater()

OtherServerData()

Detailed System Design

Classification

Subsystems:

- **Main Class**
  - Central entry point for the plugin lifecycle, runtime calls, and configuration management.
  - Responsible for managing communication across all other subsystems.
- **Game Logic Layer**
  - Contains the core mechanics of the tower defense game, such as round progression, mob spawning, tower placement, AI logic, and win/loss conditions.
- **Player Interaction Layer**
  - Facilitates communication between the system and the players, handling commands, GUI updates, and notifications.
- **Persistence Layer**
  - Manages data storage and retrieval for player saves, leaderboards, and external server communication.

Classes:

- **Main Class**
  - TowerDefensePlugin: The entry point of the plugin, responsible for lifecycle management and subsystem initialization.
- **Game Logic Layer**
  - RoundManager: Manages game rounds.
  - MobSpawner: Handles spawning of mobs.
  - TowerSpawner: Handles tower creation and placement.
  - HitDetection: Processes interactions between entities.
  - Win_LossDetection: Monitors and evaluates game state transitions.
  - TowerAI: Controls tower behaviors.
  - MobAI: Governs mob pathfinding and actions.
- **Player Interaction Layer**
  - CommandListeners: Manages player commands and triggers appropriate game logic.

    - GUIUpdater: Updates the graphical user interface based on game state changes.
- **Persistence Layer**
    - PlayerSaves: Manages saving and loading player data.
    - LeaderboardUpdater: Tracks and updates leaderboard data.
    - OtherServerData: Handles external server synchronization and data sharing.

Definition

Main:

- Serves as the entry point of the plugin. It initializes configurations, processes runtime calls, and manages the flow of information between the subsystems.

Game Logic:

- **RoundManager**
    - Controls the flow of the game by managing the start and end of rounds and scaling difficulty across rounds.
- **MobSpawner**
    - Handles the spawning of mobs at predefined or dynamic locations, ensuring the correct number and type of mobs are created per round.
- **TowerSpawner**
    - Manages the placement of towers by players, ensuring that towers are correctly initialized and interact with the game world.
- **HitDetection**
    - Detects collisions between projectiles and mobs, calculating and applying damage accordingly.
- **Win_LossDetection**
    - Evaluates game state conditions to determine whether the player has won or lost a round or the entire game.
- **TowerAI**
    - Simulates the behavior of towers, including targeting mobs, firing projectiles, and upgrading based on player input.
- **MobAI**
    - Governs mob movement and behavior, including pathfinding, attacking objectives, and responding to tower actions.

Player interaction:

- **CommandListeners**
  - Processes player commands to interact with the game, such as starting a round, upgrading towers, or viewing statistics.
- **GUIupdater**
  - Updates graphical user interfaces to reflect player status, game state, and upgrade options in real time.

Persistence:

- **PlayerSaves**
  - Manages saving and loading of player-specific data, such as tower placements, round progress, and in-game currency, ensuring continuity across sessions.
- **LeaderboardUpdater**
  - Tracks and updates player rankings or achievements based on performance, storing them for future reference or display.
- **OtherServerData**
  - Handles data exchange with external systems, such as servers or plugins, to enable cross-server functionality or shared leaderboards.

Constraints

Main class:

- **Timing**: Configuration files must be loaded during the plugin's initialization phase.
- **Storage**: Limited to JSON format for simplicity and compatibility with Minecraft server tools.
- **Constraints**:
  - Must include predefined settings.
  - Changes to configuration require a plugin reload to take effect.

Game Logic:

1. **RoundManager()**
   - **Timing**: Round progression depends on the successful completion of the current round.

- **Constraints**:
  - Only one active round can exist at a time.
  - Rounds must escalate in difficulty based on a predefined scaling system.

2. **MobSpawner()**
   - **Timing**: Mobs must spawn at precise intervals, respecting tick timing.
   - **Constraints**:
     - Spawn rates and mob types must match the current round's configuration.
     - The number of mobs spawned must not exceed server performance thresholds.
3. **TowerSpawner()**
   - **Constraints**:
     - Towers must be placed on valid blocks defined by the game.
     - Maximum tower count per player must adhere to server resource limits.
4. **HitDetection()**
   - **Constraints**:
     - Must use Minecraft's entity hitboxes for collision calculations.
     - Damage values must adhere to game balance constraints.
     - False positives must be minimized.
5. **Win_LossDetection()**
   - **Constraints**:
     - Game state must only transition to "win" or "loss" once all conditions are verified.
     - Loss detection must prioritize preventing infinite loops).
6. **TowerAI()**
   - **Constraints**:
     - Towers must prioritize the nearest mob within range.
     - Attack speed must respect predefined cooldown times.
7. **MobAI()**
   - **Constraints**:
     - Mob movement must use efficient pathfinding to prevent server lag.
     - Mobs must not deviate from the defined path unless obstructed.

Player Interaction:

- **CommandListeners()**
- **Constraints**:

    - Commands must verify player permissions before execution.
    - Invalid input must be handled gracefully.
    - Command execution must not block the main thread.
- **GUIUpdater()**
- **Constraints**:
    - GUI updates must synchronize with game state.
    - The GUI must respect Minecraft's display constraints.

Persistence Layer:

1. **PlayerSaves()**
    - **Storage**: Must store data in a lightweight, structured format.
    - **Constraints**:
        - Player data must be saved asynchronously to avoid blocking the server.
        - Save operations must ensure data integrity in case of server crashes.
2. **LeaderboardUpdater()**
    - **Timing**: Updates must occur after the end of a game or significant events.
    - **Constraints**:
        - Leaderboard updates must not overwrite existing valid data.
        - Synchronization with external leaderboard systems must handle potential latency.
3. **OtherServerData()**
    - **Constraints**:
        - Data synchronization must be non-blocking and performed in the background.
        - Network failures or desynchronization must not disrupt gameplay.

Resources

Memory:

- **Description**:
  - Managed for storing active mobs, towers, player data, and game state.
  - Each mob, tower, or player action creates temporary objects that require efficient memory usage to avoid performance degradation.
- **Potential Issues**:
  - Excessive memory usage could occur during large-scale mob spawns or extensive data storage.
- **Resolution**:
  - Optimize data structures to store only essential information.
  - Use Java's garbage collector effectively by ensuring all unused references are cleared.

Async Processes:

- **Description**:
  - Used heavily for game logic tasks like:
    - Mob AI pathfinding.
    - Tower AI targeting and firing.
    - GUI updates and player commands.
  - Computationally intensive processes run asynchronously to reduce server lag.
- **Potential Issues**:
  - High CPU usage during mob pathfinding or simultaneous tower actions.
- **Resolution**:
  - Utilize Paper API's asynchronous tasks (BukkitRunnable and runTaskLaterAsync) to offload heavy computations.
  - Limit the number of mobs or towers active at any one time.

Storage:

- **Description**:
  - Persistent storage for:
    - PlayerSaves for progress, currency, upgrades.
    - LeaderboardUpdater for rankings, achievements.
  - Storage is handled via JSON to maintain a lightweight footprint.
- **Potential Issues**:

    - Data corruption or inconsistency due to concurrent writes.
- **Resolution**:
    - Synchronize write operations to storage to prevent race conditions.
    - Use backups for critical data like player saves.

Glossary

- API – Application Programming Interface
    - Allows two different applications to interface and interact, whether it be to share, store, or transfer data and information. Necessary to interface between Spigot, Paper, and the Minecraft server
- GUI – Graphical User Interface
    - The Interface in which players interact with to perform commands for the game mode
- HotSwapAgent – Allows for a quicker development cycle by allowing Java code to updated live rather than having to restart the environment each time
- JetBrains Environment – The integrated development environment to create the Minecraft Plug-In with the language Java.
- MiniMessage – A Minecraft Plug-In making chat components of Minecraft representable by a single string.
- Paper – A type of Minecraft game server designed by the PaperMC organization that is based upon the Spigot architecture, and uses the Paper API to create a more secure, efficient way to modify Minecraft servers
- Spigot – A type of programming interface that is designed to work well with plug-ins for Minecraft servers. The architecture of which Paper is designed and laid on top of

Bibliography

HotswapProjects. *HotswapProjects/HotswapAgent: Java Unlimited redefinition of classes at runtime*. GitHub. https://github.com/HotswapProjects/HotswapAgent

JetBrains. *Download intellij idea – the leading Java and Kotlin Ide*. https://www.jetbrains.com/idea/download/?section=windows

JetBrains Marketplace. *Minecraft development - IntelliJ IDEs plugin*. https://plugins.jetbrains.com/plugin/8327-minecraft-development

Minimessages. *SPIGOTMC - High Performance Minecraft*. https://www.spigotmc.org/resources/minimessages.89382/

PaperMC Docs. *Development guide*. https://docs.papermc.io/paper/dev

SpigotMC. *Spigot plugin development*. https://www.spigotmc.org/wiki/spigot-plugin-development/